

Replanning Using Hierarchical Task Network and Operator-Based Planning

Xuemei Wang* and Steve Chien

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, M/S 525-3660, Pasadena, CA 91101-8099

Abstract. In order to scale-up to real-world problems, planning systems must be able to replan in order to deal with changes in problem context. In this paper we describe hierarchical task network and operator-based replanning techniques which allow adaptation of a previous plan to account for problems associated with executing plans in real-world domains with uncertainty, concurrency, changing objectives. We focus on replanning which preserves elements of the original plan in order to use more reliable domain knowledge and to facilitate user understanding of produced plans. We also present empirical results documenting the effectiveness of these techniques in a NASA antenna operations application.

²

Keywords: planning and reasoning, action and change, replanning, real-world application, HTN/operator-based planning

1 Introduction

As AI planning techniques move from the research environment to real-world applications, it is critical to address the needs that arise from their application environment. Specifically, many application domains are dynamic, uncertain, concurrent, and have changing objectives. Real domains may be dynamic because: the world can change independently of the plan being executed; the results of performing an action often cannot be predicted with certainty; actions and events may occur simultaneously; new goals can arise and old goals can become unimportant as time passes. In order to adapt to such context, planning systems must be able to replan, i.e., to appropriately adapt and modify the current plan to these unexpected changes in goal or state.

In this paper, we describe our replanning framework that addresses the above issues in real-world applications. This framework presumes a hybrid approach using both hierarchical task network (HTN) planning (as typified by [Erol et al 1994]) and operator-based (as typified by [Pemberthy and Weld 1992,

* Current address: Rockwell Science Center, 444 High St. Suite 400, Palo Alto, CA 94301, mei@rpal.rockwell.com

²This paper describes work performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

Carbonell et al 1992)) methods. This is a common and powerful planning architecture (such as O-Plan [Currie and Tate 1991], DPLAN [Chien et al 1995], and [Kambhampati 1995]). We present our general framework as well as its application to a real application domain,

In our work, we focus on replanning which preserves elements of the original plan instead of planning from scratch from the current state for the following reasons:

- *Domain knowledge reliability.* Encoding of domain knowledge is imperfect - domain knowledge for nominal operations scenarios is most likely to be correct. Thus, by re-using as much of the nominal operations domain knowledge as possible, the risk of encountering faulty domain knowledge is reduced.
- *Operator understanding.* The users who actually carry out the plan executions are most familiar with nominal operations and small departures from nominal operations are far easier for them to understand than novel action sequences.

In this paper, we first briefly describe the DPLAN planning framework (Section 2). We then give detailed descriptions for our replanning approach for unexpected state changes (Section 3). We present application of the replanning approach to a real-world problem, namely the Deep Space Network (DSN) Antenna Operations domain. The empirical results demonstrate the effectiveness of our replanning algorithms (Section 4). And finally we end this paper with discussions, related work, and conclusions.

2 Planning using hierarchical task network and operator-based planning

Our replanning approach presumes an integrated HTN/operator planning architecture as embodied in OPLAN ([Currie and Tate 1991]), DPLAN ([Chien et al 1995]) and [Kambhampati 1995]. In this approach, a planner can use multiple planning methods and reason about both activity-goals and state-goals. Activity-goals correspond to operational or non-operational activities and are usually manipulated using HTN planning techniques. State-goals correspond to the preconditions and effects of activity-goals, and are achieved through operator-based planning. State-goals that have not yet been achieved are also considered non-operational. Figure 1 shows the procedures used for refining these two types of goals. As soon as a refinement strategy is applied to an activity-goal or state-goal, it is removed from the list of non-operational goals. Planning is complete when all (activity) goals are operational and all (state) goals have been achieved. Further details on integrating HTN and operator-based planning paradigms is described in [Chien et al 1995].

If g is an Activity-Goal,

1. Decompose: For each decomposition rule r in R which can decompose g , apply r to produce a new plan P' . If all constraints in P' are consistent, then add P' to Q .
2. Simple Establishment: For each activity-goal g' in U that can be unified with g , simple establish g using g' and produce a new plan P' . If all constraints in P' are consistent, then add P' to Q .

If g is a State-Goal,

1. Step Addition: For each activity-goal effect that can unify with g , add that goal to P to produce a new plan P' . If the constraints in P' are consistent, then add P' to Q .
 2. Simple Establishment: For each activity-goal g' in U that has an effect e that can be unified with g , simple establish g using e and produce a new plan P' . If all constraints in P' are consistent, then add P' to Q .
-

Fig. 1. Goal Refinement Strategies

3 Replanning for unexpected state changes

This section describes our algorithm for replanning when the world changes independently of the plan being executed. The input to the replanning algorithm consists of

- the original plan being executed (*oplan*),
- a list of actions already executed (*executed-activities*), and
- the current state (*current-state*)³

Our approach for replanning assumes that (1) there is a default value for each state-goal, (2) there are well-known methods (activities) for establishing the default value for each state-goal, (3) the original plan is applicable from a state where each relevant state-goal is at its default value. These assumptions are valid in most application domains. For example, in a manufacturing domain, (1) each device (e.g. robot, clamp) has a “home” position from which the original plan executes. The home position holds default values for state-goals relevant to the device; (2) there are methods to bring each device to its home position, thus establishing the default values for the relevant state-goals; (3) each device is always at its home position at the beginning of executing the original plan.

The replanning algorithm then re-uses as much of the original plan as possible while minimizing the amount of re-execution. The replanner returns a plan consisting of the activities that need to be re-executed and those not executed, as well as the ordering constraints.

Our replanning approach proceeds as shown in the following four steps. First, the algorithm creates an activity whose effects reflect the changes to the plan

³In fact, it is not necessary to know the complete current state, as long as we have the ability to query whether a state-goal relevant to the original plan is true or false in the current state.

caused by the executed activities (Figure 3). Second, the algorithm determines the state-goals necessary for continuing execution Of the plan, but are violated due to unexpected state changes; and then applies the “reset” activities to bring each state-goal to its default value (Figure 4). Finally, the planner determiners which executed activities should be re-executed (Figure 5). This algorithm guarantees that the repaired plan re-executes all the activities that are necessary to successfully achieve the top-level goals.

Inputs: *oplan*, the original plan
 executed-activities, a list of activities already executed,
 current-state, the current state
 Output: repaired-plan, repaired plan

1. *appliedGoal* ← executeActivities(*executed-activities*, *oplan*)
 2. *Reset Goals* ← resetGoals(*oplan*, *current-state*, *executed-activities-list*)
 3. *repaired-plan* ← replan(*oplan*, *resetGoals*, *appliedGoal*)

Fig. 2. Replanning for unexpected state changes

This rest of this section gives detailed descriptions of our replanning paradigm. The descriptions employ a crucial concept, namely violated state-goal. A state-goal *sg* is violated state-goal given an original-plan (*oplan*) and a list of executed activities (*executed-activities-list*), if and only if there is a protection (protect (not *sg*)) from *g1* to *g2* in *oplan*, where *g1* is in the list of executed activities (*executed-activities-list*), while *g2* is not.

3.1 Executing activities

Figure 3 describes how the original plan is modified to reflect the changes in the plan by the executed activities. The algorithm creates an activity *appliedGoal* whose effects are the state changes caused by executing *executed-activities-list*. We assume that *executed-activities-list* is given in order of the completion of execution of each activity. In step 1, the effects and preconditions of *appliedGoal* are initialized to an empty set. In step 2, the effects of each activity in *executed-activities-list* are added to the effects of *appliedGoal* in order of their executions. An effect of an activity may overwrite the effect of a activity executed earlier. In step 3, all the activities in *executed-activities-list* are removed from the operational goal list of the plan, because in principle they should not have to be re-executed again. In step 4, all the violated state-goals are located and added back to the nonoperational goal list of the plan because they need to be re-achieved in order to ensure that the preconditions of the not executed activities are satisfied. In step 5, *appliedGoal* is promoted before all the activities that have not been executed yet.

Inputs: *executed-activities-list*, a list of activities already executed,
oplan, original plan
Output: *appliedGoal*, an activity representing the effect of executing *executed-activities-list*

1. Initialize: $Effects(appliedGoal) \leftarrow \{\}$, $Preconds(appliedGoal) \leftarrow \{\}$
2. For each activity $a \in executed-activities-list$, do:
 - For each effect $e \in Effects(a)$, do:
 - if $e \in Effect(appliedGoal)$, do nothing,
 - else if $(not\ e) \in Effect(appliedGoal)$, do:
 - $Effect(appliedGoal) \leftarrow (Effect(appliedGoal) \cup \{e\}) \setminus \{(not\ e)\}$
 - else $Effect(appliedGoal) \leftarrow Effect(appliedGoal) \cup \{e\}$
3. $OperationalGoals(oplan) \leftarrow OperationalGoals(oplan) \setminus executed-activities-list$.
4. For each effect $e \in Effects(appliedGoal)$ do:
 - if $(not\ e)$ is a **violated** state-goal, do:
 - $NonOperationalGoals(oplan) \leftarrow NonOperationalGoals(oplan) \cup (not\ e)$
5. For every activity $a \in OperationalGoals(oplan)$, do: add a after *appliedGoal*

Fig. 3. Executing activities: creating an activity *appliedGoal* to reflect the changes to the plan caused by executing the activities.

3.2 Reset state-goals

Figure 4 describes how the original plan is modified to reflect the changes in the plan caused by resetting the violated state-goals. The algorithm creates an activity *ResetGoals* for resetting the state-goals, and adds this activity into the original plan. In step 1, we initialize *resetGoals* to an activity without preconditions or effects, and insert it to the original plan. In step 2, we compute all the violated state-goals due to unexpected state changes (*violatedGoals*), i.e., those that are true in the current state but not in the expected state of applying executed activities. In step 3, we update *resetGoals* to account for the activities that establish the default values of *violatedGoals*. Since establishing the default values may result in further protection violation, in step 4 we locate all such violations and move them back to the nonoperational goals of *oplan* to reachieve them later. In step 5, *resetGoals* is promoted before all the activities that have not been executed.

3.3 Replanning

Figure 5 describes how the replanner determines which activities need to be re-executed. The algorithm analyzes the protections in the original plan, re-executing activities that are necessary to re-establish the protections that are violated by the effects of *appliedGoal* or *resetGoals*. In this algorithm, the variable *re-execute-activities* is used to store the activities that need to be re-executed, or preconditions that need to be re-achieved. This variable is used to ensure that each precondition is re-achieved at most once so that the replanner does not go into an infinite loop. In essence, the replanning algorithm recursively determines

Inputs: *current-state*, *executed-activities-list*, *oplan* Output: *oplan* including *resetGoals*

1. Initialize *resetGoals* and insert it to *oplan*
 2. *violatedGoals* \leftarrow {state-goal *g*: *g* \in *current-state*, *g* \neq *expected-state*, and *g* is a violated state-goal }
 3. For each *g* \in *ViolatedGoals*, do:
 $\text{Effects}(\text{resetGoals}) \leftarrow \text{Effects}(\text{resetGoals}) \cup \text{default-value}(g)$
 4. For each effect *e* \in *resetGoals*, do:
 if *e* is a **violated** state-goal, then
 $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \cup (\text{not } e)$
 5. For every activity *u* \in *OperationalGoals(oplan)*, do: add *u* after *resetGoals*
-

Fig. 4. Resetting goals: creating an activity *resetGoals* to reflect the changes to the plan caused by resetting the violated goals.

which executed activities are used to achieve the protections that are violated by *appliedGoal* or *resetGoals*, re-executes these activities (i.e. adds them back to the operational goals of the plan) to ensure that the preconditions of the re-executed activities and not-executed activities are achieved. During replanning, the nonoperational goals of the plan are either the preconditions of a not executed activity that are undone by *appliedGoal* or *resetGoals*, or the activities that were used to achieve these preconditions in the original plan, or the regressed preconditions of these activities. The algorithm repeatedly chooses a goal, *currentGoal*, from the nonoperational goal list, and removes it from the list (steps 1 and 2). In step 3, if *currentGoal* is a precondition of an activity, then the activity *g* that was used to achieve *currentGoal* in the original plan is added to the operational (or nonoperational) goal list of *oplan*. In addition, for any protection from an executed activity to *g*, if the protection is violated by *resetOp* or *appliedGoal*, then the protected fact is added to the nonoperational goal list because it should be re-achieved. Furthermore, if the effects of *g* violate a protection from unexecuted activity to a not yet executed activity, then the violated protection is also added to the nonoperational goals so that it can be re-achieved. In step 4, if *currentGoal* is an activity goal, then the goals that *currentGoal* decomposed into in the original plan are added back to the (non) operational goal list of *oplan*. Finally, we ensure that *resetOp* is ordered before any other activities in the repaired plan. Activities in the operational goal list form the plan returned by the replanner.

The ordering constraints of the original plans are kept as they are. Since the only ordering constraints the replanner adds to the original plan are to add the *resetGoals* before any activities need to be re-executed and any activities not yet executed, the replanning algorithm does not add any inconsistency to the original plan. Analysis of the soundness and complexity can be found in Section 5.

For example, suppose the original plan is shown in Figure 6. The protections in the plan are: (1) P1: protect *q* from A to B, (where A achieves *q*), and (2)

Inputs: *oplan*, plan with *appliedGoal* and *resetGoals* added in
Output: *repaired-plan*

```

- re-execute-a plan  $\leftarrow \text{NonOperationalGoals}(\text{oplan})$ ;
- while  $\text{NonOperationalGoals}(\text{oplan}) \neq \{\}$  do
  1. currentGoal  $\leftarrow$  choose a goal from  $\text{NonOperationalGoals}(\text{oplan})$ ;
  2.  $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \setminus \{\text{currentGoal}\}$ 
  3. if currentGoal is a precondition-goal, do:
    3.1. g  $\leftarrow$  locate the activity achieving currentGoal.
        if g  $\in$  re-execute-activities, goto 1.
    3.2. if g is an operational goal then:
         $\text{OperationalGoals}(\text{oplan}) \leftarrow \text{OperationalGoals}(\text{oplan}) \cup \{g\}$ 
        else  $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \cup \{g\}$ 
    3.3. for every protection (protect p from g0 to g), do:
        if p is deleted by resetGoals or appliedGoal, and g  $\notin$  re-execute-activities,
        then:
         $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \cup \{p\}$ ,
        re-execute-activities  $\leftarrow$  re-execute-activities  $\cup \{p\}$ 
    3.4. for every protection (protect p) from g1 to g2, and every effect e of g, such
        that p = (not e), g1  $\in$  executed-activities-list, g2  $\notin$  executed-activities-list,
        p  $\notin$  re-execute-activities, do:
         $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \cup \{p\}$ 
        re-execute-activities  $\leftarrow$  re-execute-activities  $\cup \{p\}$ 
    3.5. re-execute-activities  $\leftarrow$  re-execute-activities  $\cup \{g\}$ 
  4. if currentGoal is an activity-goal, do:
    for every child g of currentGoal,
    if g  $\notin$  re-execute-activities, then
    if g is an operational goal, then:
     $\text{OperationalGoals}(\text{oplan}) \leftarrow \text{OperationalGoals}(\text{oplan}) \cup \{g\}$ 
    else  $\text{NonOperationalGoals}(\text{oplan}) \leftarrow \text{NonOperationalGoals}(\text{oplan}) \cup \{g\}$ 
- For every activity a  $\in$   $\text{OperationalGoals}(\text{oplan})$ , do: add a after resetGoals
- return oplan

```

Fig. 5. Replanning: determining which activities need to be re-executed.

P2: protect *r* from B to D (where B achieves *r*). Suppose that when activities A, B, and C are executed, and activity D is not yet executed, an unexpected state change occurs that results in deleting *g* and *r* from the state. Then activity B needs to be re-executed because protection P2 is violated by the reset operator. Activity A also needs to be re-executed because P1 is violated by the reset operator and activity B needs to be re-executed. But activity C does not need to be re-executed. The repaired plan is shown in Figure 7.

4 Empirical Evaluation

Our replanning algorithm is a general approach which uses a domain-independent hybrid HTN/operator planning architecture. It has been tested in a real application, namely, the deep space network (DSN) antenna operation domain. This section describes the application domain, how the general replanning problem maps onto the real application domain, as well as empirical test results

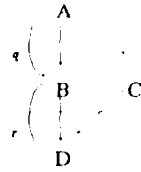


Fig. 6. An example of an original plan. Unexpected state changes occur when activities A, B, and C are executed, but D is not executed.

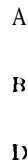


Fig. 7. Repaired plan where activities A, B are re-executed upon an unexpected state change.

4.1 Planning for Deep Space Network Antenna Operations

The Deep Space Network is a set of world-wide antenna networks which is maintained by the Jet Propulsion Laboratory (JPL). Through these antennas, JPL is responsible for providing the communications link with a multitude of spacecraft. Operations personnel are responsible for creating and maintaining this link by configuring the required antenna subsystems and performing test and calibration procedures. The task of creating the communications link is a manual and time-consuming process which requires operator input of over a hundred control directives and the constant monitoring of several dozen displays to determine the exact execution status of the system. Recently, a system called the Link Monitor and Control Operator Assistant (LMCOA), has been developed to improve operations efficiency and reduce precalibration time. The LMCOA provides semi-automated monitor and control functions to support operating DSN antennas. One of the main inputs to the LMCOA is a temporal dependency network (TDN). A TDN is a directed graph that incorporates temporal and behavioral knowledge. This graph represents the steps required to perform a communications link operation. In current operations, these TDNs are developed manually. DPLAN is an AI planning system designed to automatically generate

these TDNs based on input information describing the antenna track type and the necessary equipment configuration. DPLAN integrates HTN planning and operator-based planning. Given a set of antenna tracking goals and equipment information, DPLAN then generates a list of antenna operation steps that will create a communications link with orbiting spacecraft.

4.2 Replanning Scenarios for the DSN domain

The DSN Antenna Operations domain is dynamic, uncertain, concurrent, and has changing objectives. This domain is dynamic because the current state may unexpectedly change due to external events such as equipment (subsystem) failures. It is uncertain because actions may not always achieve their desired effects. It is concurrent because actions pertaining to different subsystems may occur simultaneously. It has changing objectives because new goals may be added after the execution of the original plan has already begun. Our replanning system is able to replan in all these scenarios. In our empirical evaluation, we focus on the first scenario, i.e., replanning when the world changes independently of the plan being executed.

Note that one alternative to replanning would be to simply reset all of the subsystems in use and completely restart the plan from scratch, re-achieving all of the desired conditions. We name this approach *complete-reset* approach. In the general case this approach is undesirable because of the *domain knowledge reliability* and *operator understanding* reasons described in Section 1. In the DSN antenna operations domain this approach is also too inefficient to be applicable for two reasons:

- *Complete-reset* approach is slow from an execution time standpoint. DSN antennas are a scarce over-subscribed resource. Completely restarting to recover from failures would reduce antenna availability for tracking purposes by increasing downtime due to time lost during recovery from changed goals, state changes, or failed actions. Additionally, delaying a track may result in lost data because it is generally infeasible to alter the spacecraft command sequence on short notice.
- *Complete-reset* approach involves resetting (powering off and then back on) all of the subsystems. This power cycling of the hardware introduces unnecessary wear on the expensive and scarce DSN subsystems.

Thus our replanning algorithm re-uses as much of the original plan as possible while minimizing the amount of re-execution by restoring the subsystem to a functioning state (generally through resetting the subsystem) and by re-achieving relevant states before continuing plan execution.

4.3 Empirical results

In Section 1, we stated that replanning by re-using the nominal plan was desirable because the domain knowledge for nominal operations is more reliable⁴. In order to verify this claim, we tested our replanning algorithm on a series of replanning problems. In these experiments, the replanning algorithm used knowledge developed for nominal operations to replan for 5 problems for each of the 3 types of replanning scenarios: subsystem failure, additional service request, and activity failure, to produce a total of 15 plans. A domain expert validated 6 plans randomly selected from the 15 repaired plans generated by our replanner. The limited size of the verified test set is due to the significant effort required to manually verify a TDN and the scarcity of the DSN domain experts. The domain expert considered all the 6 examined replans to be correct (i.e they would achieve the goals from the replan state). These results are summarized in Table 1.

# of plans	# of plans where $plan_{expert} = plan_{replanner}$	% correct	repaired plans
6	6	100%	70

Table 1. Expert judged correctness of the repaired plan

The second criterion for evaluation is that it is critical to minimize execution time of the TDNs. From the replanning point of view, this means minimizing the number of activities that need to be re-executed. For this, we compared the number of activities that are re-executed in the repaired plan versus the corresponding number in the plans generated using complete brute force replanning (i.e. resetting everything and starting from scratch). In the 15 replanning cases, the average number of activities in the original TDN is 13.8, the average number of activities executed when the failure occurs is 8.07, and the average number of activities re-executed using our replanning algorithm is 1.13. We see that the proportion of re-executed activities using our replanning algorithm is only 14% ($1.13/8.07$) of those using brute force - resetting every subsystem and starting from scratch. This demonstrates that the repaired plans generated by our replanner are significantly more efficient than brute-force replanning. Table 2 summarizes the empirical results for repaired plan efficiency.

5 Discussions

In this section, we analyze some properties of the our replanning algorithm

⁴For example, commonly there is an assumed execution context for an operator in nominal usage which is not explicitly represented in operator preconditions.

# of plans	avg. # of re-executed activities		efficiency ratio
	in <i>plan_replanner</i>	in <i>plan_from_scratch</i>	
15	1.13	8.07	14%

Table 2. Efficiency of the repaired plan

5.1 Termination and soundness

Our replanning algorithm for determining which activities require re-execution (described in Figure 5) will terminate because: (1) in the worst case when all the executed activities are added back to the operational goal list of the plan, the nonoperational goals of the plan will be empty; (2) when a violated precondition is added back to the nonoperational goal list, it takes a finite number of iterations to add the executed activity that achieves this precondition; and (3) every goal list is allowed to be added back to the (non)operational goal list at most once. Thus the complexity of the replanning algorithm is $O(n)$ where n is the length of the original plan.

Our replanning algorithms are sound assuming that the domain knowledge is correct. The soundness proof follows from analyzing the algorithm in Figure 5 by showing: (1) every possible violation of the previously achieved state is identified in the algorithm; (2) every violated state is re-established by re-executing activities in the original plan that established these conditions. Since the original plan is sound, the replanning algorithm ensures that there are no violated protections in the repaired plan, and thus is sound.

5.2 Generality

Our replanning algorithm is based on protection analysis. The protections in our plans are derived from the preconditions and effects of activities in the plan. Thus our replanning approach is applicable to all planners that maintain such protections, including [Pemberthy and Weld 1992, Carbonell et al 1992, Chien et al 1995].

We also learned that planners with only hierarchical decomposition capability are insufficient for replanning unless proper protections are specified. Most decomposition rules only specify how to decompose a high level activity to low level activities and the ordering constraints among them. Protections are not required to generate initial correct plans, although most HTN planners allow and encourage the specification of protections. In contrast, operator-based planning requires that preconditions and effects of each activity be encoded explicitly in order to function properly. Protections are generated by the planner automatically from the preconditions and effects. We believe that since protections are essential for replanning, operator-based planning is more natural for replanning purposes.

6 Related Work

One similar replanning system is the CHEF system [Hammond 1989]. In CHEF, failures are all due to unforeseen goal interactions. The CHEF system classifies a failure, infers a missing goal, and applies a critic to repair the plan. This problem differs from our replanning problem - in our case the problem is not an unrecognized goal interaction but rather a change in the problem state or goals. Thus, in our replanning problem one (inefficient) alternative is to simply re-execute the entire plan. This would be unwise in the CHEF replanning context because it is assumed the plan would simply fail again. In our replanning context, the desired outcome is to recover so as to achieve the possibly altered goal set while retaining as much of the original plan as possible.

SIPE [Wilkins 1988] also performs replanning in response to unexpected external events that change the state. SIPE first classifies the failure type and then uses this classification to apply a critic to repair the plan. Again, our replanning problem is constrained such that resetting the subsystems and re-executing the entire plan is a viable alternative - the goal is to minimize unnecessary re-execution. In SIPE's replanning scenario arbitrary replanning may be required. Thus, SIPE uses specific information in the form of critics. In our replanning problem the emphasis is on replanning to re-use the original plan, thus our approach focusses on re-establishing conditions using portions of the original plan.

Other previous work in the case-based reasoning or analogy work concentrates on adapting a case for a similar problem to the new problem situation [Velooso 1992, Karnbhampati 1990]. Their algorithms involve adding and deleting activities from the original plan based on an analysis of the applicability of the dependencies to the current problem context. This work differs from ours in two ways. First, in our approach, we handle situations where part of the plan is likely to have been executed when replanning occurs. Thus replanning must account for the altered initial state. Second, in our approach, minimizing the number of re-executed activities is desirable.

Previous work in the framework of integrating planning, executing, and replanning [Knoblock 1995, Drabble 1995] relies on the domain designer to provide repair methods for each type of failure. In the replanning problems we are addressing, it is impractical to specify a repair method for each specific class of failure. For example, in the DSN domain, there may be many different kinds of failures, failures may happen at almost any time during execution, there are tens of subsystems, etc. Hence, we have designed our algorithms to work from more general information (such as the execution status of activities). However, we still require certain specific information (e. g., the relevant subsystem to reset for an activity failure).

7 Future Work

This paper has presented a general framework for replanning required by changes in problem context. However, there are several areas for future work which are

driven by operational requirements of our target application domain of DSN antenna operations. This paper represents a first step towards tackling this complex problem and there are numerous outstanding issues which remain to be addressed. We describe several of these issues below.

In tile DSN, there is a tradeoff of the granularity of representing the activities. The activities are the lowest level primitives that the planner reasons about: each activity may contain tens of directives (commands). Sometimes during an execution failure, instead of re-executing a whole activity, it is possible to only re-execute a subset of all the directives in the activity, so that the total execution time may be shortened. To capture this plan repair knowledge, we can break an activity down to a number of activities, but then the planner must reason at a lower level of abstraction. This may result in a less maintainable knowledge base for the planner and degraded planner performance (planning speed). One area for future work is to better understand the tradeoffs and implications of selecting a particular level of abstraction for the planner.

In the DSN domain, actions take time. If the recovery actions take an extended amount of time, there may not be enough time to perform a planned equipment performance test as well as starting the acquisition of data at the required time. In this case, a tradeoff must be evaluated. For example, should the data be captured without doing the performance test? Or would the data be useless without the performance test? Endowing a planning system to reason about the utility of these differing courses of action to take the best overall course of action is a long-term goal.

In the DSN domain, during execution, some subsystems may be removed due to competing requests. Usually, these subsystems are not needed any more by the task, and are requested to be used by other tasks. What is the proper way to remove the equipment from the system? How do we unlink it with other subsystems? Enhancing the planning system to be able to reason about these types of temporal constraints and requests (using a more expressive temporal representation) is an area for future work.

Finally, in the DSN, the state of each of the subsystems is complex and contains a large amount of information. Although in principle, all relevant state information can be inferred by an expert operator, in practice this is quite difficult. How can the planning system recover from failures in a way so as to reduce the need for operators to perform complex, time-consuming and knowledge-intensive diagnoses?

8 Conclusions

In order to scale-up to real-world problems, planning systems must be able to replan in order to deal with changes in problem context. This paper has described hierarchical task network and operator-based re-planning techniques to adapt a previous plan to account for: state changes, added new goals, and failed actions. This approach attempts to preserve elements of the original plan in order to utilize *more* reliable nominal operations domain knowledge and to facilitate user understanding. In addition, the replanning methods attempt to avoid

unnecessary re-achievement of goals. We have also presented empirical results documenting the effectiveness of these techniques in a NASA antenna operations application.

References

- [Carbonell et al 1992] Carbonell, J. G.; Blythe, J.; Etzioni, C.; Gil, Y.; Joseph, R.; Kahn, D.; Knoblock, C.; Minton, S.; Pérez, M. A.; Reilly, S.; Veloso, M.; and Wang, X. PRODIGY 4.0: The Manual and Tutorial. *Technical report*, School of Computer Science, Carnegie Mellon University, 1992.
- [Chien et al 1995] S. Chien, A. Govindjee, T. Estlin, X. Wang, and R. Hill Jr., Integrating Hierarchical Task Network and Operator-based Planning Techniques to Automate Operations of Communications Antennas, *IEEE Expert*, December 1996.
- [Drabble 1995] B. Drabble. Replanning in the O(Plan) architecture, *Personal communication*, 1995.
- [Erol et al 1994] K. Erol, J. Hendler, and D. Nau, UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning, *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL, June 1994, pp. 249-254.
- [Hammond 1989] K. Hammond. Case-Based Planning: Viewing planning as a memory task. 1989.
- [Kambhampati 1990] S. Kambhampati. A theory of plan modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.
- [Kambhampati 1995] S. Kambhampati. A Comparative analysis of Partial Order Planning and Task Reduction Planning In *ACM SIGART Bulletin*, Vol.6., No.1, 1995.
- [Knoblock 1995] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of IJCAI 95*, Montreal, CA, 1995.
- [Pemberthy and Weld 1992] J. S. Pemberthy and D. S. Weld, UCPOP: A Sound Complete, Partial Order Planner for AI I., *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, October 1992.
- [Currie and Tate 1991] K. Crrrie and A. Tate, The Open Planning Architecture, In *Artificial Intelligence*, 51(1), 1991.
- [Veloso 1992] M. Veloso. Learning by Analogical Reasoning in General Problem Solving. *PhD thesis*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.
- [Wilkins 1988] D. Wilkins, Practical Planning: Extending the Classical AI Planning Paradigm. Morgan Kaufmann, 1988.